NEURAL ARCHITECTURES FOR CONTROL

Final Report

NASA/ASEE Summer Faculty Fellowship Program–1991

Johnson Space Center

Prepared by:	James K. Peterson, Ph.D.
Academic Rank:	Assistant Professor
University & Department:	Clemson University Department of Mathematical Sciences Clemson, SC 29631-1907
NASA JSC Directorate:	Information Systems
Division:	Information Technology
Branch:	Software Technology
JSC Colleague:	Robert O. Shelton, Ph.D.
Date Submitted:	July 24, 1991
Contract Number:	NGT -44-001-800

ABSTRACT

In this study, CMAC (Cerebellar Model Articulated Controller) neural architectures are shown to be viable for the purposes of real-time learning and control. Software tools for the exploration of CMAC performance are developed for three hardware platforms, the MacIntosh, the IBM PC and the SUN workstation. All algorithm development was done using the C programming language. These software tools were then used to implement an *adaptive critic* neuro-control design that learns in real-time how to back up a trailer truck. The **truck backer-upper** experiment is a standard performance measure in the neural network literature, but previously the training of the controllers was done off-line. With the CMAC neural architectures, it was possible to train the neuro-controllers on-line in real-time on a MS-DOS PC 386.

CMAC neural architectures are also used in conjunction with a hierarchical planning approach to find collision free paths over two dimensional *analog valued* obstacle fields. The method constructs a *coarse* resolution version of the original problem and then finds the corresponding *coarse* optimal path using multipass dynamic programming. CMAC artificial neural architectures are used to estimate the *analog* transition costs that dynamic programming requires. The CMAC architectures are trained in real-time for each obstacle field presented. The *coarse* optimal path is then used as a baseline for the construction of a *fine* scale optimal path through the original obstacle array.

These results are a very good indication of the potential power of the neural architectures in control design.

In order to reach as wide as audience as possible, we have run a seminar on neuro-control that has met once per week since May 20, 1991. This seminar has thoroughly discussed the CMAC architecture, relevant portions of classical control, back propagation through time and adaptive critic designs. The attendees included staff members from the Information Systems, the Engineering and Life Sciences Directorates and McDonald Douglas Corporation.

1 INTRODUCTION:

In this report, we detail our experiences with the design and application of Cerebellar Model Articulated Controller (CMAC) neural architectures to problems in path planning and control. In order to explore CMAC performance software tools using the C programming language were developed for three hardware platforms, the MacIntosh, the IBM PC and the SUN workstation. These software tools were then used to implement an *adaptive critic* neuro-control design that learns in real-time how to back up a trailer truck. The **truck backer-upper** experiment is a standard performance measure in the neural network literature, but previously the training of the controllers was done off-line. With the CMAC neural architectures, it was possible to train the neuro-controllers on-line in real-time on a MS-DOS PC 386.

CMAC neural architectures are also used in conjunction with a hierarchical planning approach to find collision free paths over two dimensional *analog valued* obstacle fields. The method constructs a *coarse* resolution version of the original problem and then finds the corresponding *coarse* optimal path using multipass dynamic programming. CMAC artificial neural architectures are used to estimate the *analog* transition costs that dynamic programming requires. The CMAC architectures are trained in real-time for each obstacle field presented. The *coarse* path is then used as a baseline in the construction of a *fine* scale path through the original obstacle array.

These results are a very good indication of the potential power of the neural architectures in control design.

In order to reach as wide as audience as possible, we ran a seminar on neuro-control that met once per week from May 20 to July 22, 1991. This seminar thoroughly discussed the CMAC architecture, relevant portions of classical control, back propagation through time and adaptive critic designs. The attendees included staff members from the Information Systems, and Life Sciences Directorates and McDonald Douglas Corporation.

2 CMAC ARCHITECTURES:

The Cerebellar Model Articulated Controller (CMAC) was first developed and described in a series of papers in the 1970's by Albus. [1, 2, 3] Their use in robotic control was further developed by Miller [6, 7, 8] in a series of papers on real time control of robotic arms. Essentially, a CMAC architecture maps the input space of the problem into a much larger **virtual address space** via what can be called **coarse encoding.** The number of entries in the virtual address space is usually quite large, perhaps 10^6 to 10^8 in number; clearly far too large to used for direct storage of tunable parameters. This large virtual address space is drastically reduced in size by *hashing* the virtual address to a smaller working address.

The input-output maps we wish the CMAC architectures to "learn" are of the form $F : [a_1, b_1] \times \ldots \times [a_N, b_N] \rightarrow R^M$. For convenience of expostion, it is easiest to describe

the CMAC architecture for a scalar valued output, ie $M = 1$; a vector-valued CMAC architecture then requires M potentially distinct CMAC structures, one for each ouput component. Let's concentrate then on the map $F : [a_1, b_1] \times \ldots \times [a_N, b_N] \rightarrow R$.

The input space for this CMAC is coarse encoded as follows. The i^{th} component of the input lives in $[a_i, b_i]$. Imagine that we have L levels of overlapping sensors that try to locate a given component in $[a_i, b_i]$. On level j, each sensor for input component i has a *receptive field width* of w_{ij}. If the sensor becomes active at position x, then it remains active until position $\min(x + w_{ij}, b_i)$. When the sensor is active, its output is 1 with value 0 everywhere outside of its region of reception. The starting points of the sensors on a given level and input component can be specified by supplying a fixed *offset*, α_{ij}, which together with the sensor field width completely determines the active region of a particular sensor. We determine the offset schedule by using an *offset base*, β_i.

$$\alpha_{ij} = (j-1)\,\beta_i\,(b_i - a_i), \quad 1 \leq i \leq L, \tag{1}$$

For example, assume there were 10 levels, the number of inputs was 16 with each input component residing in the interval was $[-.1, 1.1]$, the receptive field widths were all .60 and the offsets were all .1. This implies there are 2 sensors on level 0, active on the mutually disjoint subintervals $[0.0, 0.60)$ and $[0.60, 1.1]$; 3 sensors on level 1 active on $[0.0, 0.072)$, $[0.072, 0.672)$ and $[0.672, 1.1]$ and so forth.

Now for a given level j each component p_i of input $\vec{p}$ lies in a unique subinterval component a_j. Hence, the N inputs of $\vec{p}$ can be mapped to a N-tuple of integers $\{a_1, ..., a_N\}$. The **virtual address** of $\vec{p}$ for level j is then

$$\begin{aligned} v_j(\vec{p}) &= \sum_{j=1}^{N} a_j\, T_{j-1}, \\ T_{j-1} &= M_1\, M_2\, \ldots\, M_{j-1}. \end{aligned} \tag{2}$$

where M_j is the number of subintervals the coarse encoding provides on level j and we define $T_0 = 1$.

The virtual address therefore lies between 0 and $M_1 M_2 M_3 \ldots M_N$. This large collection of addresses is called the **virtual memory** of the CMAC architecture and it is reduced to a much smaller sized **working memory** by *hashing* the virtual addresses. We construct the working address associated with input $\vec{p}$ by computing $v_j(\vec{p})$ mod H_j, where H_j is the chosen hash size per level. The input $\vec{p}$ therefore has an associated working address for each level j, $A_j(\vec{p})$, each of which addresses one element of the finite set of weights $\{W^1_{A_1}, W^2_{A_2}, \ldots, W^L_{A_L}\}$. Thus, each input is assigned L weights and the working memory is organized into a two dimensional array of real numbers of L rows, with the j^{th} row of size H_j. The output of the CMAC corresponding to the input $\vec{p}$ is denoted by $g(\vec{p})$ and is defined by

$$g(\vec{p}) = \sum_{j=1}^{L} W^j_{A_j}. \tag{3}$$

The standard CMAC learning rule [3, 6] is then applied to the train the CMAC architectures to learn the I/O mappings. All of the CMAC architectures are initialized with zero values for the weights. Then, if d is the desired output for input $\vec{p}$), as long as the *error*, $d - g(\vec{p})$, is sufficiently large, the weights $W^j_{A_j}$ are updated using as standard delta rule,

$$\left(W^j_{A_j}\right)^{new} = \left(W^j_{A_j}\right)^{old} + \lambda\left(\frac{d - g(\vec{p})}{L}\right). \tag{4}$$

where λ is the *learning rate*, which is generally between 0 and 1. Note that, unlike standard feed forward architectures which use sigmoid transfer functions and therefore have a bounded output typically between 0 and 1, the CMAC output is obtained by *summing* values. Hence, the CMAC output does not necessarily lie between 0 and 1.

3 TRUCK BACKER-UPPER:

In this application, we will study the problem of designing a control architecture that is capable of learning in real-time to back up a standard trailer truck. This problem has become a standard benchmark for the design of self-learning control systems and the success of feed forward architectures in the solution of this problem is well documented in Nguyen and Widrow. [9] We begin with a short discussion of *adaptive control* before concluding with the truck simulation results. Standard references to the *adaptive critic* control which we use in the truck simulation include Barto [4] and Werbos. [14, 15]

3.1 Adaptive Control:

Let's consider a general control problem of the form

$$\min_{\theta \in S} \int_0^\infty f_0(x(s), \theta(s))\, ds \tag{5}$$

Subject to:

$$x'(t) = f(x(t), \theta(t)) \tag{6}$$
$$x(0) = a \tag{7}$$
$$x(t) \in \mathcal{X} \subseteq \mathcal{R}^N \tag{8}$$
$$\theta(t) \in \Theta \subseteq R^M \tag{9}$$

where x and θ are the *state vector* and *control vector* of the system, respectively; $\mathcal{S}$ is the space of functions that the control must be chosen from during the minimization process and (7) - (9) give the initialization and constraint conditions that the state and control must satisfy.

We can discretize the problem represented by equations (5) - (9) by a variety of means. For now, let's use a very simple discretization scheme; replace the differentiations by forward differences and the improper integral in (5) by a simple Riemann sum evaluated at the left-hand endpoints. We will not concern ourselves about the convergence of the resulting infinite series at this time. The discretization process leads to the following problem:

$$\min_{\theta \in \mathcal{P}(\mathcal{S})} \quad \sum_{k=0}^{\infty} f_0\left(x(k\Delta t), \theta(k\Delta t)\right)\, \Delta t \tag{10}$$

Subject to:

$$\begin{aligned} x((k+1)\Delta t) &= x(k\Delta t) + f\left(x(k\Delta t), \theta(k\Delta t)\right)\Delta t && (11)\\ x(0) &= a && (12)\\ x(k\Delta t) &\in \mathcal{X} \subseteq \mathcal{R}^{\mathcal{N}} && (13)\\ \theta(k\Delta t) &\in \Theta \subseteq R^{M} && (14) \end{aligned}$$

where Δt indicate the size of the discrete time step and the controls θ must now lie in a space of piecewise continuous functions which we will denote by $\mathcal{P}(\mathcal{S})$. We can further simplify the notation by denoting $x(k\Delta t) \equiv x_k$, $\theta(k\Delta t) \equiv \theta_k$ and $F(x,\theta) \equiv x + f(x,\theta)\,\Delta t$. If we also assume that between time $k\Delta t$ and $(k+1)\Delta t$, the control θ has only a finite number of possible actions, we can replace the set Θ by the set $\Theta_q = \{A_1, \ldots, A_q\}$. Note that the number of possible control actions q is completely independent of the number of components in the control vector M. Finally, we can think of the function value $f_0\left(x_k, \theta_k\right)\,\Delta t$ as representing some measure of the worth of our control choice at the k^{th} time step, a measure that can be used to *reinforce* our belief in the quality of our choices. Hence, we will choose the relabeling $\mathcal{R}\left(x_k, \theta_k\right) = f_0\left(x_k, \theta_k\right)\,\Delta t$ and refer to $\mathcal{R}$ as the *reinforcement function*. Also, when convenient, we will simply use the notation $\mathcal{R}_k \equiv \mathcal{R}\left(x_k, \theta_k\right)$. This leads to the more compact representation of the control problem:

$$\min_{\theta_k \in A_q} \quad \sum_{k=0}^{\infty} \mathcal{R}\left(x_k, \theta_k\right) \tag{15}$$

Subject to:

$$\begin{aligned} x_{k+1} &= F\left(x_k, \theta_k\right) && (16)\\ x(0) &= a && (17)\\ x_k &\in \mathcal{X} \subseteq \mathcal{R}^{\mathcal{N}} && (18)\\ \theta_k &\in \Theta_q && (19) \end{aligned}$$

Now assume that we have already collected knowledge of the first $L-1$ states, controls and reinforcements; hence, $x_1,\ldots,x_{L-1}$, $\theta_1,\ldots,\theta_{L-1}$ and $\mathcal{R}_1(x_1,\theta_1),\ldots,\mathcal{R}_1(x_{L-1},\theta_{L-1})$ are known. The *measure* of our performance over all *future times* due to the *control actions taken at time step k* is

$$\mathcal{U}(x_k,\theta_k) \;=\; \sum_{j=k+1}^{\infty} \mathcal{R}(x_j,\theta_j) \tag{20}$$

and we will identify $\mathcal{U}_k \equiv \mathcal{U}(x_k,\theta_k)$.

In **Adaptive-Critic** neural architectures, two separate neural networks are used together to solve the problem represented by (15) - (19). We don't know the actual value of future performance that is captured in $\mathcal{U}(x_k,\theta_k)$, so we will try to *estimate* its value using what is called a **critic** network. The correct value of the control which should be chosen to minimize performance over all future times will be estimated by another network called the **action** network.

The output of the action network will be denoted by $\mathcal{J}(W,x,\theta,\mathcal{R})$, where W indicates the parameters that need to be updated via training. We will train the critic network to approximate $\mathcal{U}_k$. A schematic of the learning algorithm for the critic network is given in table 1; i is the index for the training set, j is the index for the weight update loop and W_j are the values of the weights in the critic network after j update steps. The parameters ξ and ζ are the relative *nonnegative* weightings attached to the future prediction and the current reinforcement.

Table 1.- CRITIC LEARNING ALGORITHM

1	$W_0 = 0, j = 0$
2	$i = 1$
3	Increment j
4	Set $W_j = W_{j-1}$
5	Input $x_i, \theta_i, \mathcal{R}_i$
6	Using the previous weights, W_{j-1} Calculate the desired target $D_i = \xi\mathcal{J}(W_{j-1}, x_{i+1}, \theta_{i+1}, \mathcal{R}_{i+1}) + \zeta\mathcal{R}_i$
7	Use the delta rule of section (2) to update the CMAC weights $\mathcal{J}(W_j, x_i, \theta_i, \mathcal{R}_i) = D_i$
8	Increment i
9	If $i < L-1$ Go To (5); Else Continue
10	If $W_j \neq W_{j-1}$ Go To (2); Else Continue

It is possible to implement the learning strategy presented in table 1 into an open-ended two-cycle algorithm as follows:

Note that the algorithm in table 2 adapts the critic network indefinitely, essentialy allowing

Table 2.- REAL-TIME LEARNING ALGORITHM

1	Choose x_0, θ_0
2	Compute $x_1 = F(x_0, \theta_0)$
3	Compute $\mathcal{R}_0$
4	Train $\mathcal{J}$ so that
	$\xi \mathcal{J}(W, X_1, \theta_1, \mathcal{R}_1) + \zeta \mathcal{R}_0 = \mathcal{J}(W, x_0, \theta_0, \mathcal{R}_0)$
5	Set $x_0 = x_1, \theta_0 = \theta_1$
6	Go To 2

for as large an L as desired. After $L-1$ states, controls and reinforcements have been processed and when the critic weights have converged to W, if all target values are satisfied, we have using $\mathcal{J}_i \equiv \mathcal{J}(W, x_i, \theta_i, \mathcal{R}_i)$:

$$\mathcal{J}_i = \xi \mathcal{J}_{i+1} + \zeta \mathcal{R}_i, \ 1 \leq i \leq L-2. \tag{21}$$

Applying (21) recursively, we obtain

$$\mathcal{J}_i = \xi^{L-i-1} \mathcal{J}_{L-1} + \zeta \left(\sum_{j=0}^{L-i-2} \xi^j \mathcal{R}_{i+j} \right), \ 1 \leq i \leq L-2. \tag{22}$$

Note that is $\xi = 1$ and $\zeta = 1$, for sufficiently large L, if the infinite series given by (20) converges, $\sum_{j=0}^{L-i-2} \mathcal{R}_{i+j} \approx \mathcal{U}_{i-1}$ and we have

$$\mathcal{J}_i \approx \mathcal{J}_{L-1} + \mathcal{U}_{i-1}, \ 1 \leq i \leq L-2. \tag{23}$$

The control vectors θ that are used in the critic network are obtained by either adapting another neural architecture called the *action* network or by classical techniques. Let's assume that a neural architecture whose output is labeled $\mathcal{A}_k \equiv \mathcal{A}(V, x_k, \theta_k, \mathcal{R}_k)$ is used to predict the correct control strateg θ_{k+1} for the next time step. Hence, the predicted value of our performance measure, $\mathcal{J}_{k+1}$, suppressing the dependence on W, becomes

$$\mathcal{J}(x_{k+1}, \theta_{k+1}, \mathcal{R}(x_{k+1}, \theta_{k+1})) = \mathcal{J}(x_{k+1}, \mathcal{A}(V, x_k, \theta_k, \mathcal{R}_k), \mathcal{R}(x_{k+1}, \mathcal{A}_k). \tag{24}$$

An efficient way of updating the weights in the *action* network is to compute the rates of change of the critic network's output with respect to the weights in the action network, $\frac{\partial \mathcal{J}_{k+1}}{\partial V_j}$, for each *action* weight V_j. This can be done via back-propagation techniques.

$$\frac{\partial \mathcal{J}_{k+1}}{\partial V_j} = \left[\frac{\partial \mathcal{J}_{k+1}}{\partial \mathcal{A}_{k+1}} + \frac{\partial \mathcal{J}_{k+1}}{\partial \mathcal{R}_{k+1}}\frac{\partial \mathcal{R}_{k+1}}{\partial \mathcal{A}_{k+1}}\right]\frac{\partial \mathcal{A}_{k+1}}{\partial V_j} \tag{25}$$

Once $\frac{\partial \mathcal{J}_{k+1}}{\partial V_j}$ is available for all j, the weights of the action network at each step $k+1$ can be updated in the standard way.

$$(V_j)^{new} = (V_j)^{old} + \lambda\left(\frac{\partial \mathcal{J}_{k+1}}{\partial V_j}\right)|_{V^{old}}, \tag{26}$$

where λ is a learning rate parameter.

If a CMAC architecture is used for the action network, the output $\mathcal{J}_{k+1}$ is not a differentiable function of the inputs $\mathcal{A}_{k+1}$ and $\mathcal{R}_{k+1}$ and equations (25) and (26) can not be used to determine new values of the action network parameters. Since action networks are used in our *truck backer-upper* simulations, we have chosen to implement the control update portion of the adaptive-critic design classically as follows:

$$\theta_{k+1} = \max_{\alpha \in \Theta_q} \mathcal{J}(W, x_{k+1}, \alpha, \mathcal{R}(x_{k+1}, \alpha)) \tag{27}$$

3.2 Truck Results:

We will consider the problem of forcing a cab and two wheeled trailer to backup up along a linear trajectory from a random, sometimes jack-knifed start, while subject to small noise disturbances. We use the usual variable formulation of this truck problem: α_1 is the angle between the center-line of the cab and the x axis; α_2, the angle between the center-line of the trailer and the x axis; β, the *wheel cut angle* or the angle between the front wheel direction and the center-line of the cab; (x_c, y_c), the coordinates of the center of the front edge of the cab and (x_t, y_t), the coordinates of the center of the back edge of the trailer. The angle between the center-lines of the cab and trailer is $\tau = \pi - (\alpha_2 - \alpha_1)$ and the cab-trailer combination is considered jacknifed if $\tau < \frac{\pi}{2}$ or $\alpha_2 - \alpha_1 > \frac{\pi}{2}$.

The state variables for this problem are $\alpha_1, \alpha_2, x_c, y_c, x_t$ and y_t. We choose $\tan(\beta)$ to be the control variable. We wish to pick wheel cut angles at each time step so that the cab-trailer combination successfully tracks the given linear trajectory.

The usual truck backer-upper problem discussed in the literature, e.g., Nguyen [9], considers the problem of finding a control strategy which can successfully back up the cab-trailer from randomly positioned starts to a given position on a horizontally oriented loading dock. Feed forward architectures are trained on progressively more complicated cab-trailer movements using *back propagation through time*. The simulations are very successful, but the training phase required many thousands of runs with correspondingly heavy use of computing resources.

We have used the techniques in section 3.1 to successfully solve our cab-trailer trajectory following problem using a critic network to predict the change in the distance from (x_t, y_t) to the given trajectory with the controls chosen via (27). Our simulations learn to follow the desired trajectory in minutes using a standard MS-DOS 386 PC as the hardware platform.

4 PATH PLANNING:

In this application, we study the problem of finding the optimal path or trajectory of a autonomous device through an obstacle field for a given start and goal position. The obstacle field is modeled by a finite array of time independent analog valued pixels. It is still very difficult to solve this problem efficiently in real world applications. Various algorithms have been proposed for calculating collision free paths through obstacle fields of either fixed or moving objects. Some *fast* algorithms rely on an algorithm design that is highly parallel in nature so that fine grained multi-processor systems can be used to compute the paths quickly, e.g. Hassoun [5]. Others such as Zhu [16] solve the planning problem at multiple scales of resolution in order to quickly find a reasonable approximate path.

Here, we discuss approximate optimal paths constructed using hierarchical methods which entail constructing a *coarse* resolution version of the original obstacle array and then use multipass dynamic programming to find an optimal *coarse* path. The dynamic programming computational engine requires knowledge of the transition costs associated with moving from one node to another. The transition costs associated with the original obstacle array are easy to calculate; however, the *coarse* transition costs associated with the *coarse* obstacle array are difficult to define. They must be calculated in such a manner that the correct qualitative information about the *fine* scale path movements is not lost in the coarsening process.

We have used clustering and filtering algorithms to predict these *coarse* transition costs, Peterson, [10] and feed forward neural architectures; **binary** obstacle fields are discussed in Peterson. [11] and these results are extended in Peterson [12, 13] to calculate an estimated analog cost for a given **analog** valued directional field either via finite sums of weighted binary feedforward networks or through feed forward networks trained to assign an analog directional cost to a given analog valued directional field. However, these methods are calculationally expensive; the training in particular was fairly difficult for these data sets.

In contrast to the above work, here we model the directional costs using CMAC architectures obtaining good qualitative transition cost information with neural architectures that are trainable in a matter of minutes to 10^{-2} *or better* RMS error.

4.1 Dynamic Programming:

Assume we have an obstacle array $\mathcal{P}$ of size $n \times n$ whose value at row i and column j, p_{ij}, can take on any value between 0 and 1. We need to know the cost of moving from a given pixel to its surrounding neighbors. The *transition* cost of moving in any of the directions east (E), northeast (NE), north (N), northwest (NW), west (W), southwest (SW), south

(S) or southeast (SE) is easy to compute. If we are at interior location (i,j) in $\mathcal{P}$, we can denote the surrounding locations by $(i+a, j+b)$, for $-1 \leq a,b \leq 1$, and compute the *transition* cost of moving from position (i,j) to position $(i+a, j+b) = (i',j')$ by $T_{ij;i',j'} = \frac{1}{2}(p_{ij} + p_{i',j'})\sqrt{(a^2+b^2)}$. All transition costs that correspond to moves out of the $\mathcal{P}$ are set to infinity.

Then, given a **start node**, $S \equiv (i_s, j_s)$ and a **goal node**, $G \equiv (i_g, j_g)$, we want to find the minimum cost path through $\mathcal{P}$ from S to G, $C_{S,G,\mathcal{P}}$. *Dynamic Programming* is a technique which will calculate $C_{S,G,\mathcal{P}}$ efficiently. For our problem here, we allow movement in eight (8) directions which are divided into two distinct classes: $\Omega_1 = \{E, NE, N, SE\}$ and $\Omega_2 = \{W, NW, S, SW\}$. From our earlier notation conventions, there are associated sets of indices a and b which give rise to these direction sets, I_1 for Ω_1, and I_2 for Ω_2. Let C_{ij} denote the minimum cost of moving through $\mathcal{P}$ from node (i,j) to G with initial values of infinity with the exception of the goal cost, C_{i_g,j_g}, which is set to zero. For the set Ω_1 directions, start the calculations in the first row and last column of $\mathcal{P}$. We compute the cost values C_{ij} in the last column by moving down through the column and applying **Bellman's Principle of Optimality**, equation (28), at each node.

$$C_{ij} = \min\{C_{ij}, \min\{T_{ij;i+a,j+b} + C_{i+a,j+b} : (a,b) \in I_1\}\} \tag{28}$$

Once the last column is finished, we switch to the top of the next to the last column and move down it until finished. In this way, the cost of moving through the $\mathcal{P}$ from any position (i,j) to G is calculated and stored in a cost matrix C. At this point, no directions in Ω_2 have been used. The next set of directions is implemented similarly. This time, because the directions are essentially left and down movements, the process starts in the first column and last row position and move upwards through the first column. Upon completion of the first column, we switch to the bottom row and second column position and move up the second column. The principle of optimality is the same as given in (28) except that the index set I_1 is replaced by I_2. The **Multipass Dynamic Programming** algorithm combines the operations using the two sets of directions into an iterative procedure in the following way: *first*, compute a pass using direction set Ω_1 and *second*, compute another pass using direction set Ω_2. As long as the costs C are still changing, repeat these two passes; otherwise, stop.

4.2 Coarse Optimal Paths:

Once a random obstacle field is constructed, [12] we construct the coarse obstacle array by overlaying the fine scale array with two additional grids of size $\frac{n}{2} \times \frac{n}{2}$ and $\frac{n}{4} \times \frac{n}{4}$, respectively. The **coarse** obstacle array corresponds to the coarser of these two grids and reduces the number of active nodes in the path planning problem from n^2 to $\frac{n^2}{16}$. The intermediate mesh consists of boxes each of which contains four (4) fine scale squares. which are subdivided it into eight fine scale triangles or **sectors** labeled as shown in Figure 4.2.

Thus, each box in the intermediate grid contains eight fine scale sectors; each block in the coarse grid contains four intermediate boxes of eight fine scale sectors each for a

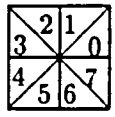

Figure 1.- Sector Subdivisions

total of thirty-two individual sectors. The **coarse** blocks can themselves be subdivided into eight sectors; each **coarse** sector contains four **fine scale** sectors. Each intermediate box can be thought of as a node in an intermediate scale path planning problem by assigning to the eight fine scale sectors within the box the pixel values of the fine scale square that they lie within. Each sector then corresponds to a triangular pixel whose value is an analog height. Further, each **coarse** sector contains four such intermediate sector values. Now each interior coarse block is surrounded by eight other coarse blocks. If we identify each block as a *coarse* node, we can use multipass dynamic programming on the smaller $\frac{n}{4} \times \frac{n}{4}$ array to find a *coarse* optimal path as long as we can find a way of assigning a transition cost for all eight of the possible directions in the sets Ω_1 and Ω_2.

For each desired direction in this "coarse" setting, we define *direction fields* as indicated in Figure 4.3 for the specific cases of the east and northeast directions. The E, N, W and S direction fields consist of 16 fine scale triangles, while the NE, NW, SW and SE direction fields consist of 24 such triangles.

In Peterson [11], these triangles are binary valued and represent an obstacle of height 1 or an empty region in the obstacle array. Hence, each fine scale triangle is modeled as an on-off event, taking the values 1 or 0. The first type of field is called a *straight* field and the second, an *angled* field. For each *straight* direction, each fine scale triangle in that direction's field is an *input*. Thus, in the **binary** case, for each *straight* direction, we need to assign to each $\vec{u} \in 2^{16}$, one of two possible outcomes; a 1 if there is a path through the particular direction field that the $\vec{u}$ represents, and a 0, otherwise. There are therefore 2^{16} possible *straight* fields for a given *straight* direction and 2^{24} possible *angled* fields for a given *angled* direction. On the other hand, in the analog case, we want to assign to each $\vec{u} \in [0,1]^{16}$ an outcome also in $[0,1]$ which represents the directional cost.

4.3 Neural Architectures:

Following Peterson [11, 12, 13], feed forward architectures were designed to "learn" both the patterns that correspond to free paths and also, the patterns that correspond to no free paths in a given direction. For the *straight* direction fields, we chose to use a feed forward network architecture with 16 input neurons, 9 hidden neurons and 1 output neuron. This architecture will be denoted a $16-9-1$ FFN for notational convenience. The *angled* direction fields were modeled using a $24-11-1$ FFN. In the **binary** case, the 0 or 1 path outcomes were determined by visual inspection for 502 examples of each type of direction field. This data was then split into 350 *training* and 152 *testing* examples or exemplars, which consist of pairs of binary direction fields and their associated binary costs.

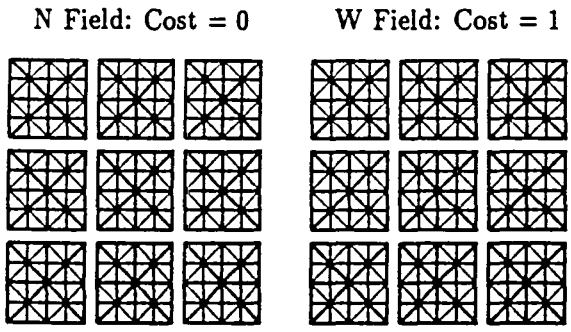

Figure 2.- Coarse Transition Cost Fields

The training results for the eight coarse transition cost feed forward networks for binary valued direction fields are discussed in Peterson. [11] These binary networks averaged 96.2% recognition on the 350 elements in the training sets and 80.0% recognition on the 152 samples in the training sets.

The cost calculations were extended to the analog case in Peterson. [12] Let $\mathcal{D}$ be a *binary* valued direction field of N pixels and let $f : \mathcal{D} \rightarrow [0,1]$ be a feed forward network of the type discussed above. Then the number of pixels, N, in each direction field is either 16 or 24 depending on whether $\mathcal{D}$ represents an *angled* or *straight* field and the network f tries to assign to each $\mathcal{D}$ a binary coarse direction cost of 0 or 1. We will use the network f to define a coarse transition cost for a direction field $\mathcal{E}$ which consists of N analog valued pixels, $p_1, \ldots, p_N$, whose values lie between 0 and 1.

From the original field $\mathcal{E}$, we can construct approximations, $\mathcal{E}^M$, as follows. For a given positive integer M, each pixel value p_i will then be in one of the mutually exclusive sets $[\frac{j-1}{M}, \frac{j}{M})$, $1 \leq j \leq M-1$ or $[\frac{M-1}{M}, 1]$. Hence, for each M, we can determine *approximate* direction fields, $\mathcal{E}^M$, where each pixel value is discretized to lie in one of M+1 values. We will label the pixel values lying in each $\mathcal{E}^M$ by p_i^M, where $p_i^M \in \left\{0, \frac{1}{M}, \ldots, \frac{M-1}{M}, 1\right\}$. We then construct *clipped* copies of the direction field $\mathcal{E}^M = \left\{p_1^{jM}, \ldots, p_N^{jM}\right\}$, where $p_i^{jM} = 0$ if $p_i^M < \frac{j}{M}$ and 1 otherwise. The *clipped* direction fields $\mathcal{E}_j^M$ are binary valued direction fields and we can compute $f(\mathcal{E}_j^M)$ for all j, $1 \leq j \leq M$. Following Peterson [12], the cost assessed for an analog valued direction field $\mathcal{E}^M$ consisting of M discrete levels of pixel values is then defined to be

$$C^M = \frac{\sum_{j=1}^{M} f(\mathcal{E}_j^M)}{M}. \tag{29}$$

For a given obstacle array, the associated coarse array contains many nodes which have well defined coarse transitions for all or some of the movement directions. The procedure outlined above permits us to use equation (29) to compute the estimated analog cost for each direction field for a given choice of gray scale M. Thus, each random obstacle array provides a wealth of training and testing data. There are then three generated paths of

interest, each of which is a matrix of 0's and 1's, with a 1 indicating that the path goes through that node and a 0, that the node is not in the path. First, $\mathcal{P}_{\mathcal{F}}$, the fine scale path; second, $\mathcal{P}_{\mathcal{FA}}$, the coarse path implied by $\mathcal{P}_{\mathcal{F}}$, where a coarse node is part of this path if at least one node in the fine scale path contained in the coarse node and third, $\mathcal{P}_{\mathcal{A}}$, the coarse scale path. The distance between $\mathcal{P}_{\mathcal{FA}}$ and $\mathcal{P}_{\mathcal{A}}$ indicates how reasonably our approximation methods work. We use the following distance measure, where (i,j) and (k,m) are the (row,column) indices of the path matrices.

$$\rho(\mathcal{P}_{\mathcal{FA}}, \mathcal{P}_{\mathcal{A}}) = \max_{\mathcal{P}_{FA}^{ij} > 0} \min_{\mathcal{P}_{A}^{km} > 0} \sqrt{((i-k)^2 + (j-m)^2)} \tag{30}$$

The directional cost information calculated using the method above generates very reasonable paths. For a 100 randomly generated 80×80 obstacle fields using gray scale $M = 20$, the associated coarse fields were 20×20 and the average $\rho(\mathcal{P}_{\mathcal{FA}}, \mathcal{P}_{\mathcal{A}})$ distance was 1.69. This indicates that a reasonable corridor can be chosen by using the coarse path as a centerline and then padding out a distance of 1 to 2 coarse blocks on either side. This removes substantial amounts of the original search space thereby lessening the computational burden of the obtaining the fine scale multipass dynamic programming solution in the corridor; at the same time, we have high confidence that we have successfully identified the correct region in the original obstacle array where the fine scale path resides.

The method outlined above provides a reasonable way to compute an estimate to the approximate analog transition costs for the obstacle avoidance problem. However, it is fairly expensive to perform the calculations suggested by equation (29). In Peterson [13], training and testing sets of analog direction fields and their associated analog costs were generated using 16 gray scales using the same feed forward architectures as in the binary case discussed above. The training results for the eight coarse transition cost feed forward networks for analog valued direction fields are summarized in Table 3. Each direction was trained using $395 - 445$ exemplars and tested on a different set of $50 - 100$ exemplars. The training and testing sets here consist of pairs of analog direction fields and their associated analog costs; the term RMS refers to the *Root Mean Square* error.

Table 3.- ANALOG TRANSITION COST FFN's

	E	W	N	S	NE	NW	SW	SE
RMS Train	.04	.07	.05	.06	.05	.05	.05	.06
RMS Test	.14	.18	.10	.17	.09	.11	.23	.10

The approximate paths generated via this method were also quite good, in spite of the relatively poor RMS errors on the testing sets. However, it was very difficult to train this data. In the rest of this work, we explore an alternative neural architecture for learning the approximate directional cost mappings which has very fast learning and sufficient generalization to also provide good performance on path generation.

4.4 CMAC Architectures:

We now use the CMAC architecture as described in 2 to estimate the directional costs required for the generation of the coarse obstacle path. The input-output maps we wish the CMAC architectures to "learn" are those discussed in section 4.3. For convenience of expostion, we will concentrate on the east direction input-output map; the input set here is a subset of R^{16} and the output set is $[0,1]$. For these experiments, the underlying direction field $\mathcal{D}$ is represented by a M gray scale approximation, which in the notation of section 4.3 is labeled $\mathcal{E}^M$. Thus, the allowable values of each component of the direction field are $\frac{j}{M}$, for $0 \leq j \leq M$. The output that is assigned to each such approximate direction field is the output calculated by the techniques given in section 4.3. This is precisely the I/O map whose training results for a $16-9-1$ *east* FFN are presented in table 3 for the case $M = 16$. The CMAC architecture for the E direction is identical to that used for the N, S and W directions; the NE, NW, SW and SE I/O maps are structured very similarly.

The input space for the east CMAC was **coarse encoded** as follows. Each component of the input lives in $[-.1, 1.1]$. The offset base β was constant for all input components and all receptive field widths were fixed at w. The offset base and sensor width used for all of the straight, E, N, W, and S, CMAC architectures and those for the angled direction fields, NW, NE, SW and SE, are given in table 4. Now pick a given direction field sample. For a given level j each component p_i of input $\vec{p}$ lies in a **unique** subinterval component a_j. Hence, the 16 inputs of $\mathcal{E}^M$ can be mapped to a 16-tuple of integers $\{a_1, ..., a_{16}\}$.

Table 4.- CMAC ARCHITECTURE

	L	β	w
Straight	10	0.1	0.60
Angled	20	0.05	0.60

We chose a uniform hash size of 1024 for all levels. Thus, each input direction field is assigned L weights and the working memory is organized into a two dimensional array of real numbers of size $L \times 1024$. The output of each CMAC and its training rule were implemented as noted in section 2. In the results below, all RMS errors on training are based on the raw CMAC outputs, but the CMAC output is clipped to lie in $[0,1]$ before being used in path planning calculations.

4.5 Training Results:

The following simulation results are from the CMAC simulation code developed at the NASA Johnson Space Center. The training samples were obtained from randomly generated 40×40 fine scale arrays with associated 10×10 coarse arrays. We obtained 90 samples per obstacle array for the straight directions and 81 samples for angled directions. We used a gray scale of $M = 20$. The CMAC architectures were trained on each of these sample sets, then a new random obstacle field was generated. This process was run 100 times, giving

a total of 9000 and 8100 potentially different training samples, however, on these runs, we typically see about 25% repeats.

The 89th training run is indicated below in table 5; all runs were limited to 10 training updates; R_0 and R_1 are the initial and final RMS errors and M_0 and M_1 are the initial and final maximum absolute errors.

Table 5.- CMAC TRAINING

	E	N	W	S	NE	NW	SW	SE
R_0	.131	.127	.115	.156	.137	.214	.197	.143
R_1	.0004	.0029	.0008	.002	.0004	.0021	.00084	.0034
M_0	.618	.51	.42	.787	.458	.753	.58	.452
M_1	.0037	.019	.0078	.014	.002	.011	.0049	.03

The procedure used above to generate training samples was then used to generate another 40 sets of directional data; for this set, all the data was saved into one file per direction. The straight directional data thus consisted of 3600 samples, 40 runs at 90 samples per run; the angled data consisted of 3240 samples, 40 runs at 81 samples per run. The performance of the trained CMAC's was then checked on these testing sets. These results are listed in table 6.

Table 6.- CMAC TESTING

	E	N	W	S	NE	NW	SW	SE
R_0	.305	.259	.387	.312	.199	.387	.261	.185
R_1	.150	.133	.154	.158	.158	.220	.20	.70
M_0	.95	.385	.95	.864	.85	.897	.950	.155
M_1	.84	.806	.91	.981	.863	1.205	.948	.909

The directional cost information calculated using the CMAC architectures also generates very reasonable paths. For a 100 randomly generated 80 × 80 obstacle fields, the associated coarse fields were 20 × 20 and the average $\rho(\mathcal{P}_{\mathcal{FA}}, \mathcal{P}_{\mathcal{A}})$ distance was 1.62. The approximate paths generated via this method were very good, in spite of the relatively poor RMS errors on the testing sets listed in table 6.

5 REFERENCES:

1. Albus, J. 1975. "A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC)." *J. Dynamic Systems, Measurement and Control*, 220- 227.

2. Albus, J. 1975. "Data Storage in the Cerebellar Model Articulation Controller (CMAC)." *J. Dynamic Systems, Measurement and Control*, 228 - 233.

3. Albus, J. 1979. "Mechanisms of Planning and Problem Solving in the Brain." *Math. Biosciences*, Vol. 45, 247 - 293.

4. Barto, A., R. Sutton and C. Anderson. 1983. "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems." *IEEE Trans. Systems, Man and Cybernetics*, Vol. SMC-13, No. 5, 834 - 846.

5. Hassoun, M. H. and A. Sanghvi. 1990. "Fast Computation of Optimal Paths in Two- and Higher Dimensional Maps", *Neural Networks*, Vol. 3: 355 - 363.

6. Miller, W. 1987. "Sensor-Based Control of Robotic Manipulators Using as General Learning Algorithm." *IEEE J. Robot. Automat.*, Vol RA-3, No. 2, 157 - 165

7. Miller, W. 1989. "Real Time Application of Neural Networks for Sensor Based Control of Robots with Vision." *IEEE Systems, Man and Cybernetics*, Vol. 19, 825 - 831.

8. Miller, W., F. Glanz and L. Kraft, III. 1990. "CMAC: An Associative Neural Network Alternative to Backpropagation." *Proceedings of the IEEE*, Vol. 78, No. 10, 1561 - 1567.

9. Nguyen, D. and B. Widrow. 1990. "Neural Networks for Self-Learning Control Systems", *IEEE Control Systems Magazine*, Vol. 10, No. 3, 18 - 23.

10. Peterson, J. 1991. "Obstacle Avoidance Using Hierarchical Dynamic Programming.", *The Proceedings of the 23rd Southeastern Symposium on System Theory*, March 1991.

11. Peterson, J. 1991. "Obstacle Avoidance Using Neural Networks and Hierarchical Dynamic Programming.",*Proceedings of the 2nd Workshop on Neural Networks: Academic/Industrial/NASA/Defense*, Society for Computer Simulation.

12. Peterson, J. 1991. "Path Planning in Analog Valued Obstacle Arrays Using Hierarchical Dynamic Programming and Neural Networks", submitted to *Artificial Neural Networks in Engineering and Science*, to be held St. Louis, Missouri, November 1991.

13. Peterson, J. 1991. "Estimating Directional Cost Information in Analog Obstacle Fields Using a Single Neural Network", submitted to *Neural Information Processing Systems*, to be held Denver, CO, December 1991.

14. Werbos, P. 1990. "Consistency of HDP Applied to a Simple Reinforcement Learning Problem", *Neural Networks*, Vol. 3, 179 - 189.

15. Werbos, P. 1990. "A Menu of Designs for Reinforcement Learning Over Time", in **Neural Networks for Control**, ed. Miller, Sutton and Werbos, 67 - 96.

16. Zhu, D. and J. Latombe. 1991. "New Heuristic Algorithms for Efficient Hierarchical Path Planning." *IEEE Trans. Robotics and Automation*, Vol. 7, No. 1, 9 - 20.